AD-A249 588

(2)

ITATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 18 Nov 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
Validation Summary Report: NATO SWG on APSE Compiler for VAX/VMS, Version VC1.82-02, VAX 8350/VMS 5.4-1 under CAIS 5.5E (Host) to VAX 8350/VMS 5.4-1 (Target), 911118I1.11236

5. FUNDING NUMBERS

6. AUTHOR(S)
IABG-AVF
Ottobrunn, Federal Republic of Germany

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
IABG-AVF, Industrieanlagen-Betriebsgeselschaft
Dept. SZT/ Einsteinstrasse 20
D-8012 Ottobrunn
FEDERAL REPUBLIC OF GERMANY

8. PERFORMING ORGANIZATION REPORT NUMBER
IABG-VSR 101

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
NATO SWG on APSE Compiler for VAX/VMS, Version VC1.82-02, VAX 8350/VMS 5.4-1 under CAIS 5.5E (Host) to VAX 8350/VMS 5.4-1 (Target), ACVC 1.11.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-11-18.

Compiler Name and Version:	**NATO SWG on APSE Compiler for VAX/VMS Version VC1.82-02**
Host Computer System:	**VAX 8350 / VMS 5.4-1 under CAIS 5.5E**
Target Computer System:	**VAX 8350 / VMS 5.4-1**

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #91111811.11236 is awarded to Alsys. This certificate expires on 01 June 1993.

This report has been reviewed and is approved.

IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

AVF Control Number: IABG-VSR 101
19 December 1991

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 911118I1.11236
NATO SWG on APSE Compiler for VAX/VMS
Version VC1.82-02
VAX 8350 / VMS 5.4-1 under CAIS 5.5E Host
VAX 8350 / VMS 5.4-1 Target

== based on TEMPLATE Version 91-05-08 ==

Prepared By:
IABG mbH, Abt. ITE
Einsteinstr. 20
W-8012 Ottobrunn
Germany

Certificate Information

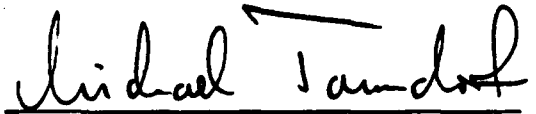
The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 91-11-18.

Compiler Name and Version:	**NATO SWG on APSE Compiler for VAX/VMS Version VC1.82-02**
Host Computer System:	**VAX 8350 / VMS 5.4-1 under CAIS 5.5E**
Target Computer System:	**VAX 8350 / VMS 5.4-1**

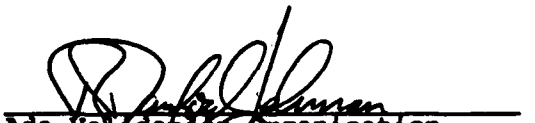
See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate #911118I1.11236 is awarded to Alsys. This certificate expires on 01 June 1993.

This report has been reviewed and is approved.



IABG, Abt. ITE
Michael Tonndorf
Einsteinstr. 20
W-8012 Ottobrunn
Germany



Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

Declaration of Conformance

Customer: Alsys GmbH & CO. KG.

Certificate Awardee: Alsys / German MoD

Ada Validation Facility: IABG mbH, Germany

ACVC Version: 1.11

Ada Implementation:

NATO SWG on APSE Compiler for VAX/VMS Version VC1.82-02

Host Computer System:
VAX 8350 / VMS Version 5.4-1 under CAIS Version 5.5E

Target Computer System: VAX 8350 / VMS Version 5.4-1

Declaration:

We, the undersigned, declare that we have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.

Rainer Hollmes — 28.11.91

Customer Signature — **Date**

Alsys GmbH & Co. KG
7500 Karlsruhe 51 Am Rüppurrer Schloß 7
Tel. 07 21/88 30 25 Fax 07 21/88 75 64

Rainer Hollmes — 28.11.91

Certificate Awardee Signature — **Date**

Alsys GmbH & Co. KG
7500 Karlsruhe 51 Am Rüppurrer Schloß 7
Tel. 07 21/88 30 25 Fax 07 21/88 75 64

DER BUNDESMINISTER DER VERTEIDIGUNG 141

Bonn, 11.Dezember 1991

Im Auftrag

Wilde

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 02 August 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	B83026B	C83026A	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

B22005A..C and B22005P (4 tests), respectively, check that control the characters SOH, STX, ETX, and DLE are illegal when outside of character literals, string literals, and comments; for this implementation those characters have a special meaning to the underlying system such that the test file is altered before being passed to the compiler. (See section 2.3.)

C24113W..Y (3 tests) contain lines of length greater than 255 characters which are not supported by this implementation.

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C41401A checks that CONSTRAINT_ERROR is raised upon the evaluation of various attribute prefixes; this implementation derives the attribute values from the subtype of the prefix at compilation time, and thus does not evaluate the prefix or raise the exception. (See Section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions; this implementation provides no package MACHINE_CODE.

The tests listed in the following table check that USE_ERROR is raised if the given file operations are not supported for the given combination of mode and access method; this implementation supports these operations.

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E CE2107G..H CE2107L CD2110B CE2110D
CE2111D CE2111H CE3111B CE3111D..E CE3114B
CE3115A

CE2108B, CE2108D, and CE3112B use the names of temporary sequential, direct, and text files that were created in other tests in order to check that the temporary files are not accessible after the completion of those tests; for this implementation, temporary files have no name.

CE2203A checks that WRITE raises USE_ERROR if the capacity of an external sequential file is exceeded; this implementation cannot restrict file capacity.

EE2401D uses instantiations of DIRECT_IO with unconstrained array and record types; this implementation raises USE_ERROR on the attempt to create a file of such types.

CE2403A checks that WRITE raises USE_ERROR if the capacity of an external direct file is exceeded; this implementation cannot restrict file capacity.

CE3304A checks that SET_LINE_LENGTH and SET_PAGE_LENGTH raise USE_ERROR if they specify an inappropriate value for the external file; there are no inappropriate values for this implementation.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST; for this implementation, the value of COUNT'LAST is greater than 150000, making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 28 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B24009A	B29001A	B38003A	B38009A	B38009B
B91001H	BC2001D	BC2001E	BC3204B	BC3205B	BC3205D

B22005A..C and B22005P (4 tests) were graded inapplicable by Evaluation Modification as directed by the AVO. These tests, respectively, check that control characters SOH, STX, ETX, and DLE are illegal outside of character literals, string literals, and comments. This implementation's underlying CAIS system gives special meaning to each of these control characters such that their effect is to alter the test files in a way that defeats the test objectives--either the characters alone, or together with any text that follows them on the line, are not passed to the compiler. Hence, B22005B and B22005P compile without error, while the other tests have syntactic errors introduced by the loss of test text.

B25002A, B26005A, and B27005A were graded passed by Evaluation Modification as directed by the AVO. These tests check that control characters SOH, STX, ETX, and DLE are illegal within of character literals, string literals, and comments, respectively. This implementation's underlying CAIS system gives special meaning to each of these control characters such that their effect is to alter the test files in the following way: these characters, and except in the case of DLE any text that follows them on the line, are not passed to the compiler. The tests were thus graded without regard for the lines that contained one of these four control characters.

C34007P and C34007S were graded passed by Evaluation Modification as directed by the AVO. These tests include a check that the evaluation of the selector "all" raises CONSTRAINT_ERROR when the value of the object is null. This implementation determines the result of the equality tests at lines 207 and 223, respectively, based on the subtype of the object; thus, the selector is not evaluated and no exception is raised, as allowed by LRM 11.6(7). The tests were graded passed given that their only output from Report.Failed was the message "NO EXCEPTION FOR NULL.ALL - 2".

C41401A was graded inapplicable by Evaluation Modification as directed by the AVO. This test checks that the evaluation of attribute prefixes that denote variables of an access type raises CONSTRAINT_ERROR when the value of the variable is null and the attribute is appropriate for an array or task type. This implementation derives the array attribute values from the subtype; thus, the prefix is not evaluated and no exception is raised, as

allowed by LRM 11.6(7), for the checks at lines 77, 87, 97, 108, 121, 131, 141, 152, 165, & 175.

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT.IDENT_INT at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

BC3204C..D and BC3205C..D (4 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests are expected to produce compilation errors, but this implementation compiles the units without error; all errors are detected at link time. This behavior is allowed by AI-00256, as the units are illegal only with respect to units that they do not depend on.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact in Germany for technical and sales information about this Ada implementation system, see:

Alsys GmbH & Co. KG
Am Rüppurrer Schloß 7
W-7500 Karlsruhe 51
Germany
Tel. +49 721 883025

Testing of this Ada implementation was conducted at the AVF's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3979	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	96	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	96	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

ACVC 1.11 was run at IABG's premises as follows: With the customer's macro parameter file the customised ACVC 1.11 was produced. Then CAIS version 5.5E as supplied by the customer was loaded and installed on the candidate VAX 8350 computer. Next the basic CAIS node model and the candidate Ada implementation were installed. Then the full set of tests was processed using command scripts provided by the customer and reviewed by the validation team. Tests were processed in two parallel CAIS sessions. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options.

Compilation was made using the following parameter settings:

```
SOURCE  => "'CURRENT_USER'DOT(SRC)"
LIBRARY => "'CURRENT_USER'ADA_LIBRARY(SAMPLE)"
LIST    => "'CURRENT_USER'DOT(LIS)"
LOG     => "'CURRENT_USER'DOT(LOG)"
```

The parameters SOURCE and LIBRARY do not have a default value and need to be specified anyway.

The default of the parameters LIST and LOG means that no listing, resp. no log output is to be produced. The values used for validation are CAIS pathnames in order to obtain the corresponding output in the file nodes specified by the respective pathnames.

Linking was made using the following parameter settings:

```
UNIT       => ...  -- Main Program to be linked
LIBRARY    => "'CURRENT_USER'ADA_LIBRARY(SAMPLE)"
EXECUTABLE => "'CURRENT_USER'DOT(EXE)"
DEBUG      =>  NO
LOG        => "'CURRENT_USER'DOT(LOG)"
```

The parameters UNIT, LIBRARY and EXECUTABLE do not have a default value and need to be specified anyway.

The default of the parameter LOG means that no log output is to be produced. The value used for validation is a CAIS pathname in order to obtain the corresponding output in the file node specified by the pathname.
The default value for the parameter DEBUG is not used, since ALSYS has provided only the runtime system which does not include debugger support.

Chapter B tests, the executable not applicable tests, and the executable tests of class E were compiled using the full listing option -l. For several tests, completer listings were added and concatenated using the option -L 'file name'. The completer is described in Appendix B, compilation system options, chapter 4.2 of the User Manual on page 39.

Test output, compiler and linker listings, and job logs were captured on a Magnetic Tape and archived at the AVF. The listings examined by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	255 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_647
$DEFAULT_MEM_SIZE	2147483648
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	VAX_VMS
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	SYSTEM.INTERRUPT_VECTOR(10)
$ENTRY_ADDRESS1	SYSTEM.INTERRUPT_VECTOR(11)
$ENTRY_ADDRESS2	SYSTEM.INTERRUPT_VECTOR(12)
$FIELD_LAST	512
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	LONG_LONG_FLOAT
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	0.0
$GREATER_THAN_DURATION_BASE_LAST	200_000.0
$GREATER_THAN_FLOAT_BASE_LAST	16#0.8#E+32
$GREATER_THAN_FLOAT_SAFE_LARGE	16#0.7FFF_FFFF_1000_000#E+32
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	16#0.7FFF_FD0#E+32
$HIGH_PRIORITY	15

```
$ILLEGAL_EXTERNAL_FILE_NAME1
                        [nodirectory]filename

$ILLEGAL_EXTERNAL_FILE_NAME2
                        FILENAME.*

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006F1.TST")

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483648

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     VMS

$LESS_THAN_DURATION     -0.0

$LESS_THAN_DURATION_BASE_FIRST
                        -200_000.0

$LINE_TERMINATOR        ' '

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        NULL;

$MACHINE_CODE_TYPE      NO_SUCH_TYPE

$MANTISSA_DOC           31

$MAX_DIGITS             33

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2_147_483_648

$MIN_INT                -2147483648

$NAME                   SHORT_SHORT_INTEGER

$NAME_LIST              VAX_VMS

$NAME_SPECIFICATION1    CAIS$DUA0:[CHAPE]X2120A.;1

$NAME_SPECIFICATION2    CAIS$DUA0:[CHAPE]X2120B.;1
```

$NAME_SPECIFICATION3	CAIS$DUA0:[CHAPE]X3119A.;1
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483648
$NEW_SYS_NAME	VAX_VMS
$PAGE_TERMINATOR	' '
$RECORD_DEFINITION	NEW INTEGER
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	10240
$TICK	0.01
$VARIABLE_ADDRESS	GET_VARIABLE_ADDRESS
$VARIABLE_ADDRESS1	GET_VARIABLE_ADDRESS1
$VARIABLE_ADDRESS2	GET_VARIABLE_ADDRESS2

APPENDIX B

COMPILATION AND LINKER SYSTEM OPTIONS

The compiler and linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

4 Compiling

After a program library has been created, one or more compilation units can be compiled in the context of this library. The compilation units must all be on the same file. One unit, a parameterless procedure, acts as the main program. If all units needed by the main program and the main program itself have been compiled successfully, they can be linked. The resulting code can then be executed by exporting the file contents to a VMS file and using the RUN command, or by using appropriate tools of the SWG APSE (CLI, Debugger), or by using CAIS operations.

§4.1 and Chapter 5 describe in detail how to call the Compiler and the Linker. In §4.2 the Completer, which is called to generate code for instances of generic units, is described.
Chapter 6 explains the information which is given if the execution of a program is abandoned due to an unhandled exception.
The information the Compiler produces and outputs in the Compiler listing is explained in §4.4.
Finally, the log of a sample session is given in Chapter 7.

4.1 Compiling Ada Units

To start the SYSTEAM Ada Compiler, use the `compile_host` command.

compile_host — **Command Description**

Format

```
PROCEDURE compile_host (

    source              : string          ;
    analyze_dependency  : yes_no_answer   := no;
    check               : yes_no_answer   := yes;
    copy_source         : yes_no_answer   := yes;
    given_by            : source_choices  := pathname;
    inline              : yes_no_answer   := yes;
    library             : pathname_type
                                  := default_library;
    list                : pathname_type   := nolist;
    log                 : pathname_type   := nolog;
    machine_code        : yes_no_answer   := no;
    optimize            : yes_no_answer   := yes;
```

```
recompile           : yes_no_answer   := no;
compiled_units      : OUT unit_descr_list  );
```

Description

The source file may contain a sequence of compilation units (cf. LRM(§10.1)). All compilation units in the source file are compiled individually. When a compilation unit is compiled successfully, the program library is updated and the Compiler continues with the compilation of the next unit on the source file. If the compilation unit contained errors, they are reported (see §4.4). In this case, no update operation is performed on the program library and all subsequent compilation units in the compilation are only analyzed without generating code.

Parameters

`source : string`
Specifies the file to be compiled. The maximum length of lines in a source file is `cais_pragmatics.text_io_columns_per_line`. The maximum number of source lines is `cais_pragmatics.text_io_lines_per_file`.

`analyze_dependency : yes_no_answer := no`
Specifies that the Compiler only performs syntactical analysis and the analysis of the dependencies on other units. The units in the source file are entered into the library if they are syntactically correct. The actual compilation is done later with the `autocompile_host` command.
Note: An already existing unit with the same name as the new one is replaced and all dependent units become obsolete, unless the source file of both are identical. In this case the library is *not* updated because the dependencies are already known.
By default, the normal, full compilation is done.

`check : yes_no_answer := yes`
Controls whether all run-time checks are suppressed. If you specify `check => no` this is equivalent to the use of PRAGMA suppress for all kinds of checks.
By default, no run-time checks are suppressed, except in cases where PRAGMA suppress_all appears in the source.

`copy_source : yes_no_answer := yes`
Controls whether a copy of the source file is kept in the library. The copy in the program library is used for later access by the Debugger or tools like the Recompiler. The name of the copy is generated by the Compiler and need normally not be known by the user. The Recompiler and the Debugger know this name. You can use the `directory_host (..., full`

=> yes, ...) command to see the file name of the copy. If a specified file contains several compilation units a copy containing only the source text of one compilation unit is stored in the library for each compilation unit. Thus the Recompiler can recompile a single unit.
If copy_source => no is specified, the Compiler only stores the name of the source file in the program library. In this case the Recompiler and the Debugger are able to use the original file if it still exists.
copy_source => yes cannot be specified together with analyze_dependency.

given_by : source_choices := pathname
given_by => pathname indicates that the string of the source parameter is to be interpreted as a pathname.
given_by => unique_identifier indicates that the string of the source parameter is to be interpreted as a unique identifier.
By default it is interpreted as a pathname.

inline : yes_no_answer := yes
Controls whether inline expansion is performed as requested by PRAGMA inline. If you specify no these pragmas are ignored.
By default, inline expansion is performed.

library : pathname_type := default_library
Specifies the program library the command works on. The compile_host command needs write access to the library.
The default is 'CURRENT_USER'ADA_LIBRARY(STD).

list : pathname_type := nolist
Controls whether a listing is written to the given file.
By default, the compile command does not produce a listing file.

log : pathname_type := nolog
Controls whether the Compiler appends additional messages onto the specified file.
By default, no additional messages are written.

machine_code : yes_no_answer := no
Controls whether machine code is appended at the listing file. machine_code has no effect if list is nolist or analyze_dependency => yes is specified.
By default, no machine code is appended at the listing file.

optimize : yes_no_answer := yes
Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.
By default, full optimization is done.

`recompile : yes_no_answer := no`
Indicates that a recompilation of a previously analyzed source is to be performed. This parameter should not be set to yes unless the command was produced by the SYSTEAM Ada Recompiler. See the `recompile_host` command.

`compiled_units : OUT unit_descr_list`
The result `compiled_units` is also a string valued item, which is the textual representation of the following list structure: The list is an unnamed possibly empty list containing a list valued item for each successfully compiled unit.
This list valued item (a unit description) is of the following structure: It is an unnamed list with three items: The first item is a string valued item containing the fully expanded name of the unit, the second item is either the string "spec", "body" or "stub", indicating whether the specification of a library unit, a body of a library unit or a subunit is named by the first item. The third item is a string valued item containing the string representation of the compilation time.

Note that the construction of this list is restricted by the following implementation specific constants of CAIS: The textual representation of a list is limited by `cais_pragmatics.list_text_length` and The length of a string item is limited by `cais_pragmatics.string_item_length`.
During compilation of a unit the compiler may detect that the description of this unit cannot be appended to the unit description list due to these limits. It then outputs an error message to the user and aborts compilation with the message:

```
+++ COMPSYS aborted because of CAIS restriction for results
```

The result `compiled_units` does not contain a description of the unit which caused the the overflow of the limit, however, the description of that unit in the library has already been updated.
When finally writing the result list the compiler may detect that the whole list of unit descriptions cannot be converted to text or the text cannot be taken as a single string item. It then outputs an error message to the user and aborts compilation with the message:

```
+++ COMPSYS aborted because of CAIS restriction for results
```

The result `compiled_units` will be empty. However, updates in the library of all successful compilations have already been performed at that time.

End of Command Description

4.2 Completing Generic Instances

Since the Compiler does not generate code for instances of generic bodies, the Completer must be used to complete such units before a program using the instances can be executed. The Completer must also be used to complete packages in the program which do not require a body. This is done implicitly when the Linker is called.

It is also possible to call the Completer explicitly with the `complete_host` command.

---

complete_host **Command Description**

Format

```
PROCEDURE complete_host (

    unit                : unitname_type       ;
    check               : yes_no_answer       := yes;
    inline              : yes_no_answer       := yes;
    library             : pathname_type
                                  := default_library;
    list                : pathname_type       := nolist;
    log                 : pathname_type       := nolog;
    machine_code        : yes_no_answer       := no;
    optimize            : yes_no_answer       := yes);
```

Description

The `complete_host` command invokes the SYSTEAM Ada Completer. The Completer generates code for all instantiations of generic units in the execution closure of the specified unit(s). It also generates code for packages without bodies (if necessary).

By default, the Completer is invoked implicitly by the `link_host` command. In normal cases there is no need to invoke it explicitly.

Parameters

`unit : unitname_type`
specifies the unit whose execution closure is to be completed.

`check : yes_no_answer := yes`
Controls whether all run-time checks are suppressed. If you specify no this is equivalent to the use of PRAGMA suppress for all kinds of checks.

By default, no run-time checks are suppressed, except in cases where PRAGMA suppress_all appears in the source.

`inline : yes_no_answer := yes`
Controls whether inline expansion is performed as requested by PRAGMA inline. If no is specified, these pragmas are ignored. By default, inline expansion is performed.

`library : pathname_type := default_library`
Specifies the program library the command works on. The `complete_host` command needs write access to the library.
The default library is `'CURRENT_USER'ADA_LIBRARY(STD)`.

`list : pathname_type := nolist`
Controls whether a listing is written to the given file.
By default, the `complete_host` command does not write a listing file.

`log : pathname_type := nolog`
Controls whether the `complete_host` command appends additional messages to the specified file.
By default, no additional messages are written.

`machine_code : yes_no_answer := no`
Controls whether a machine code listing is appended to the listing file. `machine_code => yes` has no effect if `list => nolist` is specified. By default, no machine code listing is appended to the listing file.

`optimize : yes_no_answer := yes`
Controls whether full optimization is applied in generating code. There is no way to specify that only certain optimizations are to be performed.
By default, full optimization is done.

End of Command Description

4.3 Automatic Compilation

The SYSTEAM Ada System offers three different kinds of automatic compilation. It supports

- automatic recompilation of obsolete units
- automatic compilation of modified sources
- first compilation of new sources with unknown dependencies.

In the following the term *re*compilation stands for the recompilation of an obsolete unit using the identical source which was used the last time. (This kind of recompilation could alternatively be implemented by using some appropriate intermediate representation of the obsolete unit.) This definition is stronger than that of the LRM (10.3). If a new version of the source of a unit is compiled we call it *compilation*, not a *re*compilation.

The set of units to be checked for recompilation or new compilation is described by specifying one or more units and the kind of a closure which is to be built on them. In many cases you will simply specify your main program.

The automatic recompilation of obsolete units is supported by the `recompile_host` command. It determines the set of obsolete units and generates a command file for calling the Compiler in an appropriate order. This command file is in fact an Ada program using the facilities of the package `CLI_INTERFACE` provided by the SWG APSE CLI.

The recompilation is performed using the copy of the obsolete units which is (by default) stored in the library. (If the user does not want to hold a copy of the sources the `recompile_host` command offers the facility to use the original source.)

The automatic compilation of modified sources is supported by the `autocompile_host` command. It determines the set of modified sources and generates a command file for calling the Compiler in an appropriate order. This command file is in fact an Ada program using the facilities of the package `CLI_INTERFACE` provided by the SWG APSE CLI. The basis of both the `recompile_host` and the `autocompile_host` command is the information in the library about the dependencies of the concerned units. Thus neither of these commands can handle the compilation of units which have not yet been entered in the library.

The automatic compilation of new sources is supported by the `compile_host` command together with the `analyze_dependency` parameter. This command is able to accept a set of sources in any order. It makes a syntactical analysis of the sources and determines the dependencies. The units "compiled" with this command are entered into the library, but only their names, their dependencies on other units and the name of the source files are stored in the library. Units which are entered this way can be automatically compiled using the `autocompile_host` command. They *cannot* be recompiled using the `recompile_host` command because the `recompile_host` command only recompiles units which were already compiled.

The next sections explain the usage of the `recompile_host` command, the `autocompile_host` command, and the `compile_host` command with `analyze_dependency => yes`.

4.3.1 Recompiling Obsolete Units

The recompile_host command supports the automatic recompilation of units which became obsolete because of the (re)compilation of units they depend on. The command gets as a parameter a set of units which are to be used to form the closure of units to be recompiled. The kind of the closure can be specified. The recompile_host command generates a command file with a sequence of compile commands to recompile the obsolete units which belong to the computed closure. This command file is in fact an Ada program using the facilities of the package CLI_INTERFACE provided by the SWG APSE CLI. The name of the command file can be specified using the output parameter.

recompile_host — Command Description

Format

```
PROCEDURE recompile_host (

    unit              : unitname_type       ;
    output            : pathname_type       ;
    body_ind          : yes_no_answer       := no;
    bodies_only       : yes_no_answer       := no;
    check             : yes_no_same_answer  := same;
    closure           : closure_choices     := execute;
    conditional       : yes_no_answer       := yes;
    inline            : yes_no_same_answer  := same;
    library           : pathname_type
                              := default_library;
    list              : pathname_type       := nolist;
    log               : pathname_type       := nolog;
    machine_code      : yes_no_answer       := no;
    optimize          : yes_no_same_answer  := same);
```

Description

The recompile_host command determines the specified closure based on the specified unit. Out of the units of the closure it determines the set of units which are obsolete. It generates a command file containing a compile_host (..., recompile => yes, ...) command for every obsolete unit. They are compiled in an order consistent with the WITH dependencies and the "body-of" and "subunit-of" dependencies as required by the LRM(10.3).

The recompile_host command uses the copy of the source which is stored in the library for the recompilation. By default, the compile command stores a copy of the source in the library. If there is no copy in the library - because the unit was compiled using the copy_source => no parameter - the recompile_host issues a warning and generates a compile_host command for the original source file name. It is *not* checked whether such a file still exists. This command only performs a real recompilation if the current source is the same which was last compiled.

In the command file each recompilation of a unit is executed under the condition that the recompilation of other units it depends on was successful. Thus useless recompilations are avoided. The generated command file only works correctly if the library was not modified since the command file was generated.

Note: If a unit from a parent library is obsolete it is compiled in the sublibrary in which the recompile_host command is used. In this case a later recompilation in the parent library may be hidden afterwards.

Parameters

unit : unitname_type
Specifies the unit whose closure is to be built.

output : pathname_type
Specifies the name of the generated command file.

body_ind : yes_no_answer := no
specifies that unit stands for the secondary unit with that name. By default, unit denotes the library unit. If unit specifies a subunit, the body_ind parameter need not be specified.

bodies_only : yes_no_answer := no
Controls whether all units of the closure are recompiled (default) or only the secondary units. This parameter is only effective if conditional => no is specified.

check : yes_no_same_answer := same
check => same means that the same value for the parameter check is included in the generated command file which was in effect at the last compilation. See the same parameter of the compile_host command. Otherwise the given value for the check parameter is included in the command file. By default the parameter value of the last compilation is included.

closure : closure_choices := execute
Controls the kind of the closure which is built and which is the basis for the investigation for obsolete units. closure => noclosure means that only

the specified unit is checked. `closure => compile` means that only those units on which the specified unit transitively depends are regarded. `closure => execute` means that - in addition - all related secondary units and the units they depend on are regarded. If `closure => tree` is specified, a warning is issued stating that this is not meaningful for this command and that the default value is taken instead.
By default, the execution closure is built.

`conditional : yes_no_answer := yes`
Controls whether only obsolete units are recompiled (default). `no` means that all units in the closure are recompiled whether they are obsolete or not. This parameter is useful for recompiling the complete closure with different parameters than the last time.

`inline : yes_no_same_answer := same`
`inline => same` means that the same value for the parameter `inline` is included in the generated command file which was in effect at the last compilation. See the same parameter of the `compile_host` command. Otherwise the given value for the `inline` parameter is included in the command file.
By default the parameter value of the last compilation is included.

`library : pathname_type := default_library`
Specifies the program library the command works on.
The default is `'CURRENT_USER'ADA_LIBRARY(STD)`.

`list : pathname_type := nolist`
This parameter is included in the generated command file and thus affects the generated `compile_host` command. See the same parameter with the `compile_host` command.

`log : pathname_type := nolog`
This parameter is included in the generated command file and thus affects the generated `compile_host` command. See the same parameter with the `compile_host` command.

`machine_code : yes_no_answer := no`
This parameter is included in the generated command file and thus affects the generated `compile_host` command. See the same parameter with the `compile_host` command.

`optimize : yes_no_same_answer := same`
`optimize => same` means that the same value for the parameter `optimize` is included in the generated command file which was in effect at the last compilation. See the same parameter of the `compile_host` command. Otherwise the given value for the `optimize` parameter is included in the command file.

By default the parameter value of the last compilation is included.

End of Command Description

4.3.2 Compiling New Sources

The autocompile_host command supports the automatic compilation of units for which a new source exists. The command receives as parameters a unit which is to be used to form the closure of units to be processed. The kind of closure can be specified. For every unit in the closure, the autocompile_host checks whether there exists a newer source than that which was used for the last compilation. It generates a command file with a sequence of compile_host commands to compile the units for which a newer source exists. If a unit to be compiled depends on another unit which is obsolete or which will become obsolete and for which no newer source exists, the autocompile_host command always adds an appropriate compile_host (..., recompile => yes, ...) command to make it current; the recompile parameter controls which other obsolete units are recompiled, and can indeed be used to specify that the same recompilations are done as if the recompile_host command was applied subsequently. The generated command file is in fact an Ada program using the facilities of the package CLI_INTERFACE provided by the SWG APSE CLI. The name of the command file can be specified using the output parameter.

autocompile_host Command Description

Format

```
PROCEDURE autocompile_host (

    unit              : unitname_type       ;
    output            : pathname_type       ;
    body_ind          : yes_no_answer       := no;
    bodies_only       : yes_no_answer       := no;
    check             : yes_no_same_answer  := same;
    closure           : closure_choices     := execute;
    conditional       : yes_no_answer       := yes;
    copy_source       : yes_no_answer       := yes;
    inline            : yes_no_same_answer  := same;
    library           : pathname_type
                                  := default_library;
    list              : pathname_type       := nolist;
    log               : pathname_type       := nolog;
```

```
machine_code        : yes_no_answer       := no;
optimize            : yes_no_same_answer  := same;
recompile           : recompile_choices
                              := as_necessary );
```

Description

The autocompile_host command determines the specified closure based on the specified unit. It determines the set of units for which a new source exists. The decision is based on the full file name which is stored in the library together with the modification date. If the newest version of the file has a newer modification date than the modification date which is stored in the library then the unit is said to be "new". Units which were entered with compile_host (..., analyze_dependency => yes, ...) are always said to be new.

The autocompile_host command generates a command file containing a compile_host command for every "new" unit. They are compiled in an order according to the WITH dependencies and the "body-of" and "subunit-of" relations. It is assumed that the dependencies do not change in the new sources.

Inline dependencies are not fully considered by the autocompile_host command. The autocompile_host command detects that a unit which is currently current will become obsolete because it depends on another unit (because of an inline call) which will be (re)compiled. The autocompile_host command does *not* detect that a unit u which is not current and will be (re)compiled *will* become dependent on another unit v (because of an inline call) which is currently current and will be (re)compiled too but after the compilation of u. In this case u will become obsolete again when v is (re)compiled.

When determining the compilation order the autocompile_host command tries to choose the same order as last time by considering the compilation dates in the library, where possible. This strategy should solve the inline problem in most cases.

In the generated command file each compilation of a unit is executed under the condition that the compilations of other units it depends on were successful. Thus useless compilations are avoided. The generated command file only works correctly if the library has not been modified since the command file was generated.

The autocompile_host command does not fully handle the problem which arises when several compilation units are contained within one source file; it only avoids the multiple compilation of the same source file. If you want

to use the autocompile_host command it is recommended not to keep several compilation units in one source.

Parameters

unit : unitname_type
Specifies the unit whose closure is to be built.

output : pathname_type
Specifies the name of the generated command file.

body_ind : yes_no_answer := no
specifies that unit stands for the secondary unit with that name. By default, unit denotes the library unit. If unit specifies a subunit, the body_ind parameter need not be specified.

bodies_only : yes_no_answer := no
Controls whether all new units of the closure are compiled (default) or only the secondary units. This parameter is only effective if conditional => no is specified.

check : yes_no_same_answer := same
check => same means that the same value for the parameter check is included in the generated command file which was in effect at the last compilation. See the same parameter of the compile_host command. Otherwise the given value for the check parameter is included in the command file. By default the parameter value of the last compilation is included.

closure : closure_choices := execute
Controls the kind of the closure which is built and which is the basis for the investigation for new sources. closure => noclosure means that only the specified units are checked. closure => compile means that only those units on which the specified unit(s) transitively depend(s) are regarded. closure => execute means that - in addition - all related secondary units and the units they depend on are regarded. If closure => tree is specified, a warning is issued stating that this is not meaningful for this command and that the default value is taken instead.
By default, the execution closure is investigated for new sources.

conditional : yes_no_answer := yes
Controls whether the check for new sources is performed (default). no means that all units in the closure are compiled disregarding the modification date. This parameter is useful for compiling the complete closure with different parameters than the last time.

copy_source : yes_no_answer := yes

This parameter is included in the generated command file and thus affects the generated `compile_host` command. See the same parameter with the `compile_host` command. This parameter has no effect for the recompilation of obsolete units in accordance with the `recompile_host` command where `copy_source => yes` cannot be specified.

`inline : yes_no_same_answer := same`
`inline => same` means that the same value for the parameter `inline` is included in the generated command file which was in effect at the last compilation. See the same parameter of the `compile_host` command. Otherwise the given value for the `inline` parameter is included in the command file.
By default the parameter value of the last compilation is included.

`library : pathname_type := default_library`
Specifies the program library the command works on. The `autocompile_host` command needs read access to the library. For executing the generated command file you need write access.
The default is `'CURRENT_USER'ADA_LIBRARY(STD)`.

`list : pathname_type := nolist`
This parameter is included in the generated command file and thus affects the generated `compile_host` command. See the same parameter with the `compile_host` command.

`log : pathname_type := nolog`
This parameter is included in the generated command file and thus affects the generated `compile_host` command. See the same parameter with the `compile_host` command.

`machine_code : yes_no_answer := no`
This parameter is included in the generated command file and thus affects the generated `compile_host` command. See the same parameter with the `compile_host` command.

`optimize : yes_no_same_answer := same`
`optimize => same` means that the same value for the parameter `optimize` is included in the generated command file which was in effect at the last compilation. See the same parameter of the `compile_host` command. Otherwise the given value for the `optimize` parameter is included in the command file.
By default the parameter value of the last compilation is included.

`recompile : recompile_choices := as_necessary`
Controls whether the `autocompile_host` command additionally recompiles obsolete units. With `recompile => as_necessary` only those units are recompiled which are obsolete or become obsolete *and* are used by other units

which are to be compiled because of new sources. `recompile => same_status` additionally recompiles those units of the considered closure which will become obsolete during the compilation of new sources. This option specifies that there shall not be more obsolete units after the execution of the command file than before. `recompile => as_possible` specifies that all obsolete units of the closure and all units which will become obsolete are recompiled. This is equivalent to a subsequent call of the `recompile_host` command after the run of the command file generated by the `autocompile_host` command.

End of Command Description

4.3.3 First compilation

The SYSTEAM Ada System supports the first compilation of sources for which no compilation order is known by the `compile_host` command with parameter `analyze_dependency` in combination with the `autocompile_host` command.

With the `analyze_dependency` parameter the Compiler accepts sources in any order and performs the syntax analysis. If the sources are syntactically correct the units which are defined by the sources are entered into the library. Their names, their dependencies on other units and the name of the source files are stored in the library. Units which are entered this way can be automatically compiled using the `autocompile_host` command, i.e. the Autocompiler computes the first compilation order for the new sources. The name of the main program, of course, must be known and specified with the `autocompile_host` command.

Note that the `compile_host (..., analyze_dependency => yes, ...)` command replaces other units in the library with the same name as a new one. Thus the library may be modified even if the new units contain semantic errors; but the errors will not be detected until the command file generated by the `autocompile_host` command is run. Hence it is recommended to use an empty sublibrary if you do not know anything about the set of new sources.

If there are several sources containing units with the same name the last analyzed one will be kept in the library.

The `autocompile_host` command issues special warnings if the information about the new units is incomplete or inconsistent.

4.4 Compiler Listing

By default, messages of the Compiler are listed on `'standard_output`. A complete listing can be obtained on a file by using the `list` parameter with the `compile_host`, `complete_host` or `link_host` command. The generated listing file(s) will contain the whole source together with the messages of the Compiler/Completer.

The listing for a compilation unit starts with the kind and the name of the current unit.

Example:

```
= PROCEDURE  MAIN
```

The format effectors ASCII.HT, ASCII.VT, ASCII.CR, ASCII.LF and ASCII.FF are represented by a '~' character in the listing. In any case, those source lines which are included in the listing are numbered to make locating them in the source file easy.

Errors are classified into SYMBOL ERROR, SYNTAX ERROR, SEMANTIC ERROR, RESTRICTION, COMPILER ERROR, WARNING and INFORMATION:

SYMBOL ERROR
: pinpoints an inappropriate lexical element. "Inappropriate" can mean "inappropriate in the given context". For example, '2' is a lexical element of Ada, but it is not appropriate in the literal 2#012#.

SYNTAX ERROR
: indicates a violation of the Ada syntax rules as given in the LRM(Appendix E).

SEMANTIC ERROR
: indicates a violation of Ada language rules other than the syntax rules.

RESTRICTION
: indicates a restriction of this implementation. Examples are representation clauses which are provided by the language but are not supported in this implementation; or situations in which the internal storage capacity of the Compiler for some sort of entity is exceeded.

COMPILER ERROR
: We hope you will never see a message of this sort.

WARNING
: messages tell the user facts which are likely to cause errors (for example, the raising of exceptions) at runtime.

INFORMATION
: messages tell the user facts which may be useful to know but probably do not endanger the correct running of the program; for example, that a library unit named in a context clause is not used in the current compilation unit.

Warnings and information messages have no influence on the success of a compilation. If there are any other diagnostic messages, the compilation was unsuccessful.

All error messages are self-explanatory. If a source line contains errors, the error messages for that source line are printed immediately below it. The exact position in the source to which an error message refers is marked by a number. This number is also used to relate different error messages given for one line to their respective source positions.

In order to enable semantic analysis to be carried out even if a program is syntactically incorrect, the Compiler corrects syntax errors automatically by inserting or deleting symbols. The source positions of insertions/deletions are marked with a vertical bar and a number. The number has the same meaning as above. If a larger region of the source text is affected by a syntax correction, this region is located for the user by repeating the number and the vertical bar at the end as well, with dots in between these bracketing markings.

A complete Compiler listing follows which shows the most common kinds of error messages, the technique for marking affected regions and the numbering scheme for relating error messages to source positions. It is slightly modified so that it fits into the page width of this document:

```
**************************************************************************
**                                                                      **
** SYSTEAM ADA - COMPILER        VAX/VMS/CAIS x VAX/VMS  1.82          **
**                                                                      **
** 90-01-29/08:39:44                                                    **
**                                                                      **
**************************************************************************

==========================================================================
=                                                                        =
=                                    Started at  : 08:39:44              =
=                                                                        =
=                                                                        =
=  PROCEDURE  LISTING_EXAMPLE                                            =
=                                                                        =
     1        PROCEDURE listing_example IS
     2        abc : procedure integer RANGE 0 .. 9 := 10E-1;
                    |1.....1|
                                                          1
>>>>>  SYNTAX ERROR
         Symbol(s)  deleted (1)
>>>>>  SYMBOL ERROR (1)    An exponent for an integer literal must not
                           have a minus sign
     3        def integer RANGE 0 .. 9;
```

```
                |1
>>>>>  SYNTAX ERROR
         Symbol(s) inserted (1):  :
     4        bool : boolean;
     5        BEGIN
     6        bool := (abc AND (def * 1)) OR adf;
                      1      2            3
>>>>>  SEMANTIC ERROR (1)  Actual parameter for LEFT is not of
                           appropriate type
>>>>>  SEMANTIC ERROR (2)  Actual parameter for RIGHT is not of
                           appropriate type
>>>>>  SEMANTIC ERROR (3)  Identifier ADF not declared
     7        END;
=                                                                    =
=  PROCEDURE  LISTING_EXAMPLE                                        =
=                                                                    =
=  **** Number of Errors              :     6                        =
=  **** Number of Warnings            :     0                        =
=                                                                    =
=  **** Number of Source  Lines       :      7                       =
=  **** Number of Comment Lines       :      0                       =
=  **** Number of Lexical Elements    :    42                        =
=  **** Code Size in Bytes            :     0                        =
=  **** Number of Diana Nodes created :    51                        =
=  **** Symbol  Error  in Line        :      2.                      =
=  **** Syntax  Error  in Line        :      2,     3.               =
=  **** Semantic Error in Line        :      6.                      =
=  **** CPU Time used                 :     1.4  Seconds             =
=                                      Finished at : 08:39:50        =
=                                                                    =
======================================================================
**********************************************************************
**                                                                  **
** End of  Ada  Compilation           CPU Time used :  1.4  Seconds **
**                                                                  **
**********************************************************************
```

5 Linking

An Ada program is a collection of units used by a main program which controls the execution. The main program must be a parameterless library procedure; any parameterless library procedure within a program library can be used as a main program.

The VAX/VMS system linker is used by the SYSTEAM Ada Linker.

To link a program, call the `link_host` command.

link_host — Command Description

Format

```
PROCEDURE link_host (

    unit                : unitname_type      ;
    executable          : pathname_type      ;
    check               : yes_no_answer      := yes;
    complete            : yes_no_answer      := yes;
    debug               : yes_no_answer      := yes;
    external            : string             := "";
    inline              : yes_no_answer      := yes;
    library             : pathname_type
                                  := default_library;
    linker_options      : string             := "";
    linker_listing      : pathname_type      := nolist;
    list                : pathname_type      := nolist;
    log                 : pathname_type      := nolog;
    machine_code        : yes_no_answer      := no;
    map                 : pathname_type      := nomap;
    optimize            : yes_no_answer      := yes;
    -- not used:
    relocatable         : yes_no_answer      := no;
    selective           : yes_no_answer      := no);
```

Description

The `link_host` command invokes the SYSTEAM Ada Linker.

The Linker builds an executable image which can be started by exporting the file contents to a VMS file and using the RUN command or by using

appropriate tools of the SWG APSE (CLI, Debugger) or by using CAIS operations directly, see Chapter 6.

Parameters

`unit : unitname_type`
Specifies the library unit which is the main program. This must be a parameterless library procedure.

`executable : pathname_type`
Specifies the name of the executable image. The corresponding node is expected to exist and its kind should be the predefined kind for executable images.

`check : yes_no_answer := yes`
This parameter is passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command.

`complete : yes_no_answer := yes`
Controls whether the Completer of the SYSTEAM Ada System is invoked before the linking is performed. Only specify `complete => no` if you are sure that there are no instantiations or implicit package bodies to be compiled, e.g. if you repeat the `link_host` command with different linker options.

`debug : yes_no_answer := yes`
Controls whether debug information for the SWG APSE Debugger is to be generated and included in the executable image. If the program shall run under the control of the Debugger it must be linked with the `debug => yes` parameter.
By default, debug information is included in the executable image.

`external : string := ""`
The specified string is directly passed to the VMS linker. It serves to supply the names of external object files or libraries and must be of the form

```
string = { file-spec [ /LIBRARY ] [ /INCLUDE ] } , ...
```

Example:

```
link_host (..., external => "a.obj, b.olb/LIBRARY", ...)
```

`inline : yes_no_answer := yes`
This parameter is passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command.

`library : pathname_type := default_library`
Specifies the program library the command works on. The `link_host` command needs write access to the library unless `complete => no` is specified. If `complete => no` is specified, the `link_host` command needs only read access. The default library is `'CURRENT_USER'ADA_LIBRARY(STD)`.

`linker_options : string := ""`
The specified string is directly passed to the VMS linker. It serves to supply global VMS linker parameters for special purposes and must be of the following form:

```
string = { /vms_linker_parameter } ...
```

Note that you must enclose this string in quotes because it contains parameters to be interpreted by the VMS linker. You must not use the `/MAP` and `/EXECUTABLE` parameters here. Use instead the respective parameters of the `link_host` command.

Example:

```
link_host ("main", ..., linker_options => "/CONTIGUOUS", ...)
```

`linker_listing : pathname_type := nolist`
Unless `linker_listing => nolist` is specified, the Linker of the SYSTEAM Ada System produces a listing in the given file containing a table of symbols which are used for linking the Ada units. This table is helpful when debugging an Ada program with the VMS debugger.
By default, the Linker does not produce a listing.

`list : pathname_type := nolist`
This parameter is passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command.

`log : pathname_type := nolog`
This parameter controls whether the command writes additional messages onto the specified file, and is also passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command.

`machine_code : yes_no_answer := no`
This parameter is passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command. If `linker_listing` is unequal `nolist` and `machine_code => yes` is specified, the Linker of the SYSTEAM Ada System appends a listing with the machine code of the program starter to the file given by `linker_listing`. The program starter is a routine which contains the calls of the necessary elaboration routines and a call for the Ada subprogram which is the main program.

By default, no machine code listing is generated.

`map : pathname_type := nomap`
Specifies whether the map listing of the VMS linker is to be preserved in the specified file.

`optimize : yes_no_answer := yes`
This parameter is passed to the implicitly invoked Completer. See the same parameter with the `complete_host` command.

`relocatable : yes_no_answer := no`
This parameter is not considered by this SYSTEAM Ada System

`selective : yes_no_answer := no`
Controls selective linking. Selective linking means that only the code of those subprograms which can actually be called is included in the executable image. By default, the code of all subprograms of all packages in the execution closure of the main procedure is linked into the executable image.

Note: The code of the runtime system and of the predefined units is always linked selectively.

End of Command Description

The VMS Linker is called by the SYSTEAM Ada Linker through the means provided by the package CAIS_HOST_SPECIFIC_SERVICES, see the CAIS User's Manual for details. results in

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are contained in the following Predefined Language Enviroment (chapter 13 page 97 ff of the compiler user manual).

13 Predefined Language Environment

The predefined language environment comprises the package standard, the language-defined library units and the implementation-defined library units.

13.1 The Package STANDARD

The specification of the package standard is outlined here; it contains all predefined identifiers of the implementation.

```
PACKAGE standard IS

  TYPE boolean IS (false, true);

  -- The predefined relational operators for this type are as follows:

  -- FUNCTION "="   (left, right : boolean) RETURN boolean;
  -- FUNCTION "/="  (left, right : boolean) RETURN boolean;
  -- FUNCTION "<"   (left, right : boolean) RETURN boolean;
  -- FUNCTION "<="  (left, right : boolean) RETURN boolean;
  -- FUNCTION ">"   (left, right : boolean) RETURN boolean;
  -- FUNCTION ">="  (left, right : boolean) RETURN boolean;

  -- The predefined logical operators and the predefined logical
  -- negation operator are as follows:

  -- FUNCTION "AND" (left, right : boolean) RETURN boolean;
  -- FUNCTION "OR"  (left, right : boolean) RETURN boolean;
  -- FUNCTION "XOR" (left, right : boolean) RETURN boolean;

  -- FUNCTION "NOT" (right : boolean) RETURN boolean;

  -- The universal type universal_integer is predefined.

  TYPE integer IS RANGE - 2_147_483_648 .. 2_147_483_647;

  -- The predefined operators for this type are as follows:

  -- FUNCTION "="  (left, right : integer) RETURN boolean;
  -- FUNCTION "/=" (left, right : integer) RETURN boolean;
  -- FUNCTION "<"  (left, right : integer) RETURN boolean;
```

```
-- FUNCTION "<=" (left, right : integer) RETURN boolean;
-- FUNCTION ">"  (left, right : integer) RETURN boolean;
-- FUNCTION ">=" (left, right : integer) RETURN boolean;

-- FUNCTION "+"   (right : integer) RETURN integer;
-- FUNCTION "-"   (right : integer) RETURN integer;
-- FUNCTION "ABS" (right : integer) RETURN integer;

-- FUNCTION "+"   (left, right : integer) RETURN integer;
-- FUNCTION "-"   (left, right : integer) RETURN integer;
-- FUNCTION "*"   (left, right : integer) RETURN integer;
-- FUNCTION "/"   (left, right : integer) RETURN integer;
-- FUNCTION "REM" (left, right : integer) RETURN integer;
-- FUNCTION "MOD" (left, right : integer) RETURN integer;

-- FUNCTION "**"  (left : integer; right : integer) RETURN integer;

-- An implementation may provide additional predefined integer types.
-- It is recommended that  the names of such additional types end
-- with INTEGER as in SHORT_INTEGER  or  LONG_INTEGER.  The
-- specification of each operator for the type universal_integer, or
-- for any additional predefined integer type, is obtained by
-- replacing INTEGER  by the name of the type in the specification
-- of the corresponding operator of the type INTEGER, except for the
-- right operand of the exponentiating operator.

TYPE short_integer IS RANGE - 32_768 .. 32_767;

TYPE short_short_integer IS RANGE - 128 .. 127;

-- The universal type universal_real is predefined.

TYPE float IS DIGITS 9 RANGE
        - 16#0.7FFF_FFFF_FFFF_FF8#E+32 ..
          16#0.7FFF_FFFF_FFFF_FF8#E+32;
-- the corresponding machine type is D-FLOAT

-- The predefined operators for this type are as follows:

-- FUNCTION "="   (left, right : float) RETURN boolean;
-- FUNCTION "/="  (left, right : float) RETURN boolean;
-- FUNCTION "<"   (left, right : float) RETURN boolean;
-- FUNCTION "<="  (left, right : float) RETURN boolean;
-- FUNCTION ">"   (left, right : float) RETURN boolean;
-- FUNCTION ">="  (left, right : float) RETURN boolean;

-- FUNCTION "+"   (right : float) RETURN float;
```

```
-- FUNCTION "-"   (right : float) RETURN float;
-- FUNCTION "ABS" (right : float) RETURN float;

-- FUNCTION "+"   (left, right : float) RETURN float;
-- FUNCTION "-"   (left, right : float) RETURN float;
-- FUNCTION "*"   (left, right : float) RETURN float;
-- FUNCTION "/"   (left, right : float) RETURN float;

-- FUNCTION "**"  (left : float; right : integer) RETURN float;

-- An implementation may provide additional predefined floating
-- point types. It is recommended that the names of such additional
-- types end with FLOAT as in SHORT_FLOAT or LONG_FLOAT.
-- The specification of each operator for the type  universal_real,
-- or for any additional predefined floating point type, is obtained
-- by replacing FLOAT by the name of the type in the specification of
-- the corresponding operator of the type FLOAT.

TYPE short_float IS DIGITS 6 RANGE
        - 16#0.7FFF_FF8#E+32 .. 16#0.7FFF_FF8#E+32;
-- the corresponding machine type is F-FLOAT

TYPE long_float IS DIGITS 15 RANGE
        - 16#0.7FFF_FFFF_FFFF_FC#E+256 ..
          16#0.7FFF_FFFF_FFFF_FC#E+256;
-- the corresponding machine type is G-FLOAT

TYPE long_long_float IS DIGITS 33 RANGE
        - 16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#E+4096 ..
          16#0.7FFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_C#E+4096;
-- the corresponding machine type is H-FLOAT

-- In addition, the following operators are predefined for universal
-- types:

-- FUNCTION "*" (left : UNIVERSAL_INTEGER; right : UNIVERSAL_REAL)
                RETURN UNIVERSAL_REAL;
-- FUNCTION "*" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
                RETURN UNIVERSAL_REAL;
-- FUNCTION "/" (left : UNIVERSAL_REAL;    right : UNIVERSAL_INTEGER)
                RETURN UNIVERSAL_REAL;

-- The type universal_fixed is predefined.
-- The only operators declared for this type are

-- FUNCTION "*" (left : ANY_FIXED_POINT_TYPE;
                right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;
```

```
-- FUNCTION "/" (left : ANY_FIXED_POINT_TYPE;
                 right : ANY_FIXED_POINT_TYPE) RETURN UNIVERSAL_FIXED;

-- The following characters form the standard ASCII character set.
-- Character  literals corresponding to control characters are not
-- identifiers.

TYPE character IS
     (nul,  soh,  stx,  etx,     eot,  enq,  ack,  bel,
      bs,   ht,   lf,   vt,      ff,   cr,   so,   si,
      dle,  dc1,  dc2,  dc3,     dc4,  nak,  syn,  etb,
      can,  em,   sub,  esc,     fs,   gs,   rs,   us,
      ' ',  '!',  '"',  '#',     '$',  '%',  '&',  ''',
      '(',  ')',  '*',  '+',     ',',  '-',  '.',  '/',
      '0',  '1',  '2',  '3',     '4',  '5',  '6',  '7',
      '8',  '9',  ':',  ';',     '<',  '=',  '>',  '?',
      '@',  'A',  'B',  'C',     'D',  'E',  'F',  'G',
      'H',  'I',  'J',  'K',     'L',  'M',  'N',  'O',
      'P',  'Q',  'R',  'S',     'T',  'U',  'V',  'W',
      'X',  'Y',  'Z',  '[',     '\',  ']',  '^',  '_',
      '`',  'a',  'b',  'c',     'd',  'e',  'f',  'g',
      'h',  'i',  'j',  'k',     'l',  'm',  'n',  'o',
      'p',  'q',  'r',  's',     't',  'u',  'v',  'w',
      'x',  'y',  'z',  '{',     '|',  '}',  '~',  del);

FOR character USE  --  128 ascii CHARACTER SET WITHOUT HOLES
          (0, 1, 2, 3, 4, 5, ...., 125, 126, 127);

-- The predefined operators for the type CHARACTER are the same as
-- for any enumeration type.

PACKAGE ascii IS
  --  Control characters:
  nul : CONSTANT character := nul;   soh : CONSTANT character := soh;
  stx : CONSTANT character := stx;   etx : CONSTANT character := etx;
  eot : CONSTANT character := eot;   enq : CONSTANT character := enq;
  ack : CONSTANT character := ack;   bel : CONSTANT character := bel;
  bs  : CONSTANT character := bs;    ht  : CONSTANT character := ht;
  lf  : CONSTANT character := lf;    vt  : CONSTANT character := vt;
  ff  : CONSTANT character := ff;    cr  : CONSTANT character := cr;
  so  : CONSTANT character := so;    si  : CONSTANT character := si;
  dle : CONSTANT character := dle;   dc1 : CONSTANT character := dc1;
  dc2 : CONSTANT character := dc2;   dc3 : CONSTANT character := dc3;
  dc4 : CONSTANT character := dc4;   nak : CONSTANT character := nak;
  syn : CONSTANT character := syn;   etb : CONSTANT character := etb;
  can : CONSTANT character := can;   em  : CONSTANT character := em;
  sub : CONSTANT character := sub;   esc : CONSTANT character := esc;
```

```
   fs  : CONSTANT character := fs;    gs  : CONSTANT character := gs;
   rs  : CONSTANT character := rs;    us  : CONSTANT character := us;
   del : CONSTANT character := del;

   -- Other characters:

   exclam     : CONSTANT character := '!';
   quotation  : CONSTANT character := '"';
   sharp      : CONSTANT character := '#';
   dollar     : CONSTANT character := '$';
   percent    : CONSTANT character := '%';
   ampersand  : CONSTANT character := '&';
   colon      : CONSTANT character := ':';
   semicolon  : CONSTANT character := ';';
   query      : CONSTANT character := '?';
   at_sign    : CONSTANT character := '@';
   l_bracket  : CONSTANT character := '[';
   back_slash : CONSTANT character := '\';
   r_bracket  : CONSTANT character := ']';
   circumflex : CONSTANT character := '^';
   underline  : CONSTANT character := '_';
   grave      : CONSTANT character := '`';
   l_brace    : CONSTANT character := '{';
   bar        : CONSTANT character := '|';
   r_brace    : CONSTANT character := '}';
   tilde      : CONSTANT character := '~';

   lc_a : CONSTANT character := 'a';
   ...
   lc_z : CONSTANT character := 'z';

END ascii;

-- Predefined subtypes:

SUBTYPE natural  IS integer RANGE 0 .. integer'last;
SUBTYPE positive IS integer RANGE 1 .. integer'last;

-- Predefined string type:

TYPE string IS ARRAY(positive RANGE <>) OF character;

PRAGMA pack(string);

-- The predefined operators for this type are as follows:

-- FUNCTION "="  (left, right : string) RETURN boolean;
```

```
   -- FUNCTION "/=" (left, right : string) RETURN boolean;
   -- FUNCTION "<"  (left, right : string) RETURN boolean;
   -- FUNCTION "<=" (left, right : string) RETURN boolean;
   -- FUNCTION ">"  (left, right : string) RETURN boolean;
   -- FUNCTION ">=" (left, right : string) RETURN boolean;

   -- FUNCTION "&" (left : string;    right : string)    RETURN string;
   -- FUNCTION "&" (left : character; right : string)    RETURN string;
   -- FUNCTION "&" (left : string;    right : character) RETURN string;
   -- FUNCTION "&" (left : character; right : character) RETURN string;

   TYPE duration IS DELTA 2#1.0#E-14 RANGE
           - 131_072.0 .. 131_071.999_938_964_843_75;

   -- The predefined operators for the type DURATION are the same
   -- as for any fixed point type.

   -- the predefined exceptions:

   constraint_error : EXCEPTION;
   numeric_error    : EXCEPTION;
   program_error    : EXCEPTION;
   storage_error    : EXCEPTION;
   tasking_error    : EXCEPTION;

END standard;
```

13.2 Language-Defined Library Units

The following language-defined library units are included in the master library:

The package system
The package calendar
The generic procedure unchecked_deallocation
The generic function unchecked_conversion
The package io_exceptions
The generic package sequential_io
The generic package direct_io
The package text_io
The package low_level_io

13.3 Implementation-Defined Library Units

The master library also contains the implementation-defined library units `collection_manager` and `timing`.

13.3.1 The Package COLLECTION_MANAGER

In addition to unchecked storage deallocation (cf. LRM(§13.10.1)), this implementation provides the generic package `collection_manager`, which has advantages over unchecked deallocation in some applications; e.g. it makes it possible to clear a collection with a single reset operation. See §15.10 for further information on the use of the collection manager and unchecked deallocation.

The package specification is:

```
GENERIC
   TYPE elem IS LIMITED PRIVATE;
   TYPE acc  IS ACCESS elem;
PACKAGE collection_manager IS

   TYPE status IS LIMITED PRIVATE;

   PROCEDURE mark (s : OUT status);

      -- Marks the heap of type ACC and
      -- delivers the actual status of this heap.

   PROCEDURE release (s : IN status);

      -- Restore the status s on the collection of ACC.
      -- RELEASE without previous MARK raises CONSTRAINT_ERROR

   PROCEDURE reset;

      -- Deallocate all objects on the heap of ACC

   PRIVATE
      -- private declarations
```

```
END collection_manager;
```

A call of the procedure release with an actual parameter s causes the storage occupied by those objects of type acc which were allocated after the call of mark that delivered s as result, to be reclaimed. A call of reset causes the storage occupied by all objects of type acc which have been allocated so far to be reclaimed and cancels the effect of all previous calls of mark.

See §15.2.1 for information on static and dynamic collections and the attribute STORAGE_SIZE.

13.3.2 The Package TIMING

The package timing provides a facility for CPU-time measurement. The package specification is:

```
PACKAGE timing IS

   FUNCTION  cpu_time RETURN natural;

   timing_error : EXCEPTION;

END timing;
```

A call of the function cpu_time returns the CPU-time consumed by the running process in milliseconds. The value natural'last will be reached after 24 days, 20 hours, 31 minutes, 23 seconds and 647 milliseconds.
The exception timing_error will be raised if a constraint_error or numeric_error occurs within cpu_time.

Note that for CAIS based processes a CAIS service is available to request the consumed time. The package provided here uses facilities of VMS to perform the task.

15 Appendix F

This chapter, together with the Chapters 16 and 17, is the Appendix F required in the LRM, in which all implementation-dependent characteristics of an Ada implementation are described.

15.1 Implementation-Dependent Pragmas

The form, allowed places, and effect of every implementation-dependent pragma is stated in this section.

15.1.1 Predefined Language Pragmas

The form and allowed places of the following pragmas are defined by the language; their effect is (at least partly) implementation-dependent and stated here.

CONTROLLED
has no effect.

ELABORATE
is fully implemented. The SYSTEAM Ada System assumes a PRAGMA elaborate, i.e. stores a unit in the library as if a PRAGMA elaborate for a unit u was given, if the compiled unit contains an instantiation of u (or for a generic program unit in u) and if it is clear that u *must* have been elaborated before the compiled unit. In this case an appropriate information message is given. By this means it is avoided that an elaboration order is chosen which would lead to a PROGRAM_ERROR when elaborating the instantiation.

INLINE
Inline expansion of subprograms is supported with the following restrictions: the subprogram must not contain declarations of other subprograms, tasks, generic units or body stubs. If the subprogram is called recursively only the outer call of this subprogram will be expanded.

INTERFACE

is supported for ASSEMBLER and VMS. `PRAGMA interface(assembler,...)` provides an interface with the internal calling conventions of the SYSTEAM Ada System. See §15.1.3 for further description.

`PRAGMA interface(VMS,...)` is provided to support the VAX procedure calling standard. §15.1.4 describes how to use this pragma.

`PRAGMA interface` should always be used in connection with the `PRAGMA external_name` (see §15.1.2), otherwise the Compiler will generate an internal name that leads to an unsolved reference during linking. These generated names are prefixed with an underline; therefore the user should not use names beginning with an underline.

LIST

is fully implemented. Note that a listing is only generated when the `list` option is specified accordingly with the `compile_host` (or `complete_host` or `link_host`) command.

MEMORY_SIZE

has no effect.

OPTIMIZE

has no effect; but see also the `optimize` parameter with the `compile_host` command, §4.1

PACK

see §16.1.

PAGE

is fully implemented. Note that form feed characters in the source do not cause a new page in the listing. They are - as well the other format effectors (horizontal tabulation, vertical tabulation, carriage return, and line feed) - replaced by a " character in the listing.

PRIORITY

There are two implementation-defined aspects of this pragma: First, the range of

the subtype priority, and second, the effect on scheduling (Chapter 14) of not giving this pragma for a task or main program. The range of subtype priority is 0 .. 15, as declared in the predefined library package system (see §15.3); and the effect on scheduling of leaving the priority of a task or main program undefined by not giving PRAGMA priority for it is the same as if the PRAGMA priority 0 had been given (i.e. the task has the lowest priority).

SHARED
is fully supported.

STORAGE_UNIT
has no effect.

SUPPRESS
has no effect, but see §15.1.2 for the implementation-defined PRAGMA suppress_all.

SYSTEM_NAME
has no effect.

15.1.2 Implementation-Defined Pragmas

BYTE_PACK
see §16.1.

EXTERNAL_NAME (<string>, <ada_name>)
<ada_name> specifies the name of a subprogram or of an object declared in a library package, <string> must be a string literal. It defines the external name of the specified item. The Compiler uses a symbol with this name in the call instruction for the subprogram. The subprogram declaration of <ada_name> must precede this pragma. If several subprograms with the same name satisfy this requirement the pragma refers to that subprogram which is declared last.
Upper and lower cases are distinguished within <string>, i.e. <string> must be given exactly as it is to be used by external routines. This pragma will be used in

connection with the pragmas `interface (vms)` or `interface (assembler)` (see §15.1.1).

RESIDENT (<ada_name>)

this pragma causes the value of the object to be held in memory and prevents assignments of a value to the object <ada_name> from being eliminated by the optimizer (see §4.1) of the SYSTEAM Ada Compiler. The following code sequence demonstrates the intended usage of the pragma:

```
...
x : integer;
a : SYSTEM.address;
   ...
BEGIN
  x := 5;
  a := x'ADDRESS;
  do_something (a);  -- let do_something be a non-local
                     -- procedure
                     -- a.ALL will be read in the body
                     -- of do_something
  x := 6;
  ...
```

If this code sequence is compiled by the SYSTEAM Ada Compiler with `optimize => yes` the statement `x := 5;` will be eliminated because from the point of view of the optimizer the value of x is not used before the next assignment to x. Therefore

```
PRAGMA resident (x);
```

should be inserted after the declaration of x.

This pragma can be applied to all those kinds of objects for which the address clause is supported (cf. §16.5).
It will often be used in connection with the PRAGMA `interface (vms, ... )` (see §15.1.4).

SUPPRESS_ALL

causes all the runtime checks described in the LRM(§11.7) to be suppressed; this pragma is only allowed at the start of a compilation before the first compilation unit; it applies to the whole compilation.

15.1.3 Pragma Interface (Assembler,...)

This section describes the internal calling conventions of the SYSTEAM Ada System, which are the same ones which are used for subprograms for which a PRAGMA `interface (ASSEMBLER,...)` is given. Thus the actual meaning of this pragma is simply that the body needs and must not be provided in Ada, but in object form using the `external` parameter with the `link_host` command.

> In many cases it is more convenient to follow the VAX procedure calling standard. Therefore the SYSTEAM Ada System provides the PRAGMA `interface(VMS,...)`, which supports the standard return of the function result and the standard register saving. This pragma is described in the next section.

The internal calling conventions are explained in four steps:

- Parameter passing mechanism
- Ordering of parameters
- Type mapping
- Saving registers

Parameter passing mechanism:

The parameters of a call to a subprogram are placed by the caller in an area called *parameter block*. This area is aligned on a longword boundary and contains parameter values (for parameter of scalar types), descriptors (for parameter of composite types) and alignment gaps.
For a function subprogram an extra field is assigned at the beginning of the parameter block containing the function result upon return. Thus the return value of a function is treated like an anonymous parameter of mode OUT. No special treatment is required for a function result except for return values of an unconstrained array type (see below).

A subprogram is called using the **CALLG** instruction. The address pointing to the beginning of the parameter block and the code address of the subprogram are specified as operands. Within the called subprogram the parameter block is accessed via the argument pointer **AP**.

In general, the ordering of the parameter values within the parameter block does not agree with the order specified in the Ada subprogram specification. When determining the position of a parameter within the parameter block the calling mechanism and the size and alignment requirements of the parameter type are considered. The size and alignment requirements and the passing mechanism are described in the following:

Scalar parameters or parameters of access types are passed by value, i.e. the values of the actual parameters of modes IN or IN OUT are copied into the parameter block

before the call. Then, after the subprogram has returned, values of the actual parameters of modes IN OUT and OUT are copied out of the parameter block into the associated actual parameters. The parameters are aligned within the parameter block according to their size: A parameter with a size of 8, 16 or 32 bits (or a multiple of 8 bits greater than 32) has an alignment of 1, 2 or 4 (which means that the object is aligned to a byte, word or longword boundary within the parameter block). If the size of the parameter is not a multiple of 8 bits (which may be achieved by attaching a size specification to the parameter's type in case of an integer, enumeration or fixed point type) it will be byte aligned. Parameters of access types are always aligned to a longword boundary.

For parameters of composite types, descriptors are placed in the parameter block instead of the complete object values. A descriptor contains the address of the actual parameter object and, possibly, further information dependent on the specific parameter type. The following composite parameter types are distinguished:

- A parameter of a constrained array type is passed by reference for all parameter modes.
- For a parameter of an unconstrained array type, the descriptor consists of the address of the actual array parameter followed by the bounds for each index range in the array (i.e. FIRST(1), LAST(1), FIRST(2), LAST(2), ...). The space allocated for the bound elements in the descriptor depends on the type of the index constraint.
- For functions whose return value is an unconstrained array type a descriptor for the array is passed in the parameter block as for parameters of mode OUT. The fields for its address and all array index bounds are filled up by the function before it returns. In contrast to the procedure for an OUT parameter, the function allocates the array in its own stack space. The function then returns without releasing its stack space. After the function has returned, the calling routine copies the array into its own memory space and then deallocates the stack memory of the function.
- A constrained record parameter is passed by reference for all parameter modes.
- For an unconstrained record parameter of mode IN, the parameter is passed by reference using the address pointing to the record.
  If the parameter has mode OUT or IN OUT, the value of the `CONSTRAINED` attribute applied to the actual parameter is passed as an additional boolean IN parameter (which occupies one byte in the parameter block and is aligned to a byte boundary). The boolean IN parameter and the address are treated like two consecutive parameters in a subprogram specification, i.e. the positions of the two parameters within the parameter block are determined independently of each other.

For all kinds of composite parameter types the pointer pointing to the actual parameter object is represented by a 32 bit address, which is always aligned to a longword boundary.

Ordering of parameters:

The ordering of the parameters in the parameter block is determined as follows:

The parameters are processed in the order they are defined in the Ada subprogram specification. For a function the return value is treated as an anonymous parameter of mode OUT at the start of the parameter list. Because of the size and alignment requirements of a parameter it is not always possible to place parameters in such a way that two consecutive parameters are densely located in the parameter block. In such a situation a gap, i.e. a piece of memory space which is not associated with a parameter, exists between two adjacent parameters. Consequently, the size of the parameter block will be larger than the sum of the sizes used for all parameters. In order to minimize the size of the gaps in a parameter block an attempt is made to fill each gap with a parameter that occurs later in the parameter list. If during the allocation of space within the parameter block a parameter is encountered whose size and alignment fit the characteristics of an available gap, then this gap is allocated for the parameter instead of appending it at the end of the parameter block. As each parameter will be aligned to a byte, word or longword boundary the size of any gap may be one, two or three bytes. Every gap of size three bytes can be treated as two gaps, one of size one byte with an alignment of 1 and one of size two bytes with an alignment of 2. So, if a parameter of size two is to be allocated, a two byte gap, if available, is filled up. A parameter of size one will fill a one byte gap. If none exists but a two byte gap is available, this is used as two one byte gaps. By this first fit algorithm all parameters are processed in the order they occur in the Ada program.

A called subprogram accesses each parameter for reading or writing using the argument pointer AP incremented by an offset from the start of the parameter block suitable for the parameter. So the value of a parameter of a scalar type or an access type is read (or written) directly from (into) the parameter block. For a parameter of a composite type the actual parameter value is accessed via the descriptor stored in the parameter block which contains a pointer to the actual object.

Type mapping:

To access individual components of array or record types, knowledge about the type mapping for array and record types is required. An array is stored as a sequential concatenation of all its components. Normally, pad bits are used to fill each component to a byte, word, longword or a multiple thereof in dependence on the size and alignment requirements of the components' subtype. This padding may be influenced using one of the PRAGMAs pack or byte_pack (cf. §16.1). The offset of an individual array component is then obtained by multiplying the padded size of one array component by the number of components stored in the array before it. This number may be determined from the number of elements for each dimension using the fact that the array elements are stored row by row. (For unconstrained arrays the number of elements for each dimension can be found in the descriptor stored in the parameter block.)

A record object is implemented as a concatenation of its components. Initially, locations are reserved for those components that have a component clause applied to them. Then locations for all other components are reserved. Any gaps large enough to hold components without component clauses are filled, so in general the record components are rearranged. Components in record variants are overlaid. The ordering mechanism of the components within a record is in principle the same as that for ordering the parameters in the parameter block.

A record may hold implementation-dependent components (cf. §16.4). For a record component whose size depends on discriminants, a generated component holds the offset of the record component within the record object. If a record type includes variant parts there may be a generated component (cf. §16.4) holding the size of the record object. This size component is allocated as the first component within the record object if this location is not reserved by a component clause. Since the mapping of record types is rather complex you should introduce record component clauses for each record component if you want to pass an object of that type to a non Ada subprogram to be sure to access the components correctly.

Saving registers:

The last aspect of the calling conventions discussed here is that of saving registers. The calling subprogram assumes that the values of the registers R0 .. R5 will be destroyed by the called subprogram and saves them of its own accord. If the called subprogram wants to modify further register these must be specified in the procedure entry mask to ensure that they are pushed on the stack when entering the subprogram and restored when leaving it. This differs from the VAX procedure calling standard, which demands that all registers which are modified by the subprogram except R0 and R1 must be specified in the mask.

15.1.4 Pragma Interface(VMS,...)

The SYSTEAM Ada System supports `PRAGMA interface(VMS,...)`.

With the help of this pragma *and* by obeying some rules (described below) subprograms can be called which follow the VAX procedure calling standard. As the user must know something about the internal calling conventions of the SYSTEAM Ada System we recommend reading §15.1.3 before reading this section and before using `PRAGMA interface(VMS,...)`.

In comparison to `PRAGMA interface(assembler,...)` `PRAGMA interface(VMS,...)` has two additional effects:

- If the subprogram is a function it moves the function result in R0 (or R0 and R1 for results of size 8 byte) into the allocated area at the beginning of the parameter block.

- It controls register allocation in such a way that it is assumed (according to the standard) that the called subprogram only modifies the registers R0 and R1.

The code for subprograms for which PRAGMA Interface(VMS,...) is specified can reside in one of the VMS object libraries, which are searched by default, or can be provided using the external parameter with the link_host command.

In order to follow the VAX procedure calling standard in all aspects the user has to cover three other aspects:

- The order of the parameters in the parameter block must be the same as in the Ada procedure specification. Because of the gap optimization - as described in the previous section - you can enforce this by using only parameters of 4 byte length (for example integer or access types). According to the VAX procedure calling standard shorter parameters are not used anyway. Note that the Compiler will use a shorter representation where possible, even if there is no PRAGMA pack and no a representation clause. Thus a parameter of type boolean cannot be passed by value; you must specify integer instead, and pass the appropriate corresponding integer value (0 or 1). Most of the parameters of the VMS system calls or of the run time library are passed by reference anyway. In this case the parameter can be declared to be of type SYSTEM.ADDRESS and the actual parameter is then <variable>'ADDRESS. For the function result the same rules apply as for the parameters, with one exception: You can also use the (predefined) types FLOAT and LONG_FLOAT, which both have a size of 8 bytes (or equivalent floating point types which are mapped onto these two predefined types.)

- It is an additional convention that the first parameter in the parameter block is of type integer and denotes the number of parameters which are passed. This is easily achieved by the user by adding a first parameter of type integer with a default value. By this means, the calls are not affected, provided parameter association is always named and not positional.

  Note: PRAGMA interface(vms,...) adds an additional parameter in front of this if the subprogram is a function. The Compiler then correctly passes the address in the parameter block where the first Ada parameter begins, i.e the first extra parameter for the function result, which exists only in the SYSTEAM Ada System internal calling conventions, is not present in the called function.

- The last aspect which is not handled by the SYSTEAM Ada System is the support of complex types, such as string descriptors, etc. These types can be built in the Ada program by appropriate record declarations, with representation clauses if necessary.

Summary: If you use only 4 byte parameters and specifiy PRAGMA interface(VMS,...) and add an additional first integer parameter which holds the number of parameters, then the procedure call generated by the SYSTEAM Ada System conforms to the VAX procedure calling standard.

Note: For parameters which are passed by reference by declaring the formal parameter of type SYSTEM.ADDRESS and passing the parameter by <object>'ADDRESS, the SYSTEAM Ada System typically needs PRAGMA **resident** to be specified for the actual parameter. This prevents the Optimizer of the SYSTEAM Ada System from holding the value of the parameter in a register or even eliminating it completely, because there is no further access to the parameter - from the Optimizers point of view. (cf. §15.1.2)

The SYSTEAM Ada System does not check the observance of the VAX procedure calling standard. If it is violated the call of the non Ada routine will be erroneous.

The following example shows the intended usage of PRAGMA **interface (VMS,...)** to call a VMS system routine. First some types with representation specifications and objects of those types are declared. Then the Ada specification of **sys_qio** appears and is related through appropriate pragmas to the system service SYS$QIO. It is assumed that the function is called in the body of the main program. This example further shows the use of interrupt entries with the SYSTEAM Ada System (cf. §16.5.1.2).

```
WITH system, text_io;

PROCEDURE vms_routine IS

   readprompt     : CONSTANT := 55;
   m_noecho       : CONSTANT := 64;
   null_address   : CONSTANT system.address := system.address_zero;

   TYPE iosb_type IS
      RECORD
         condition_value : short_integer;
         transfer_count  : short_integer;
         dev_spec_info   : integer;
      END RECORD;

   FOR iosb_type USE
      RECORD
         condition_value AT 0 RANGE 0  .. 15;
         transfer_count  AT 0 RANGE 16 .. 31;
         dev_spec_info   AT 4 RANGE 0  .. 31;
      END RECORD;

   chan           : integer;                  -- channel number
   res            : integer;                  -- return code

   io_buffer      : string (1 .. 80);
```

```
   PRAGMA resident (io_buffer);

   prompt_buffer : string (1 .. 2) := "* ";
   PRAGMA resident (prompt_buffer);

   iosb          : iosb_type;
   PRAGMA resident (iosb);

   FUNCTION  sys_qio (n      : integer := 12;  -- number of parameters
                      efn    : integer := 0;   -- event flag number
                      chan   : integer;        -- channel
                      func   : integer;        -- function code
                      iosb   : system.address := null_address;
                                               -- IO status block
                      astadr : system.address := null_address;
                                               -- AST routine
                      astprm : integer := 0;
                      p1     : system.address; -- IO buffer
                      p2     : integer := 0;   -- IO buffer length
                      p3     : integer := 0;   -- timeout
                      p4     : integer := 0;   -- read terminator
                      p5     : system.address := null_address;
                                               -- prompt buffer
                      p6     : integer := 0)   -- prompt buffer length
                RETURN integer;

   PRAGMA interface (vms, sys_qio);
-- and additionally
   PRAGMA external_name ("SYS$QIO", sys_qio);

   vector_number : CONSTANT := 0;
   ast_parameter : CONSTANT := 123_456_789;

   TASK buffer_handler IS
      PRAGMA priority (15);

      ENTRY buffer_read (param : integer);
      FOR buffer_read USE AT system.interrupt_vector (vector_number);
   END buffer_handler;

   TASK BODY buffer_handler IS
   BEGIN
      ACCEPT buffer_read (param : integer) DO
         text_io.put_line (io_buffer);
      END buffer_read;

   END buffer_handler;
```

```
BEGIN

   -- after an appropriate ASSIGN channel call:

   res :=
      sys_qio (chan => chan,
               func => readprompt + m_noecho,
               iosb => iosb'address,
               astadr => system.ast_service (vector_number),
               astprm => ast_parameter,
               p1 => io_buffer'address,
               p2 => io_buffer'length,
               p3 => 0,
               p4 => 0,
               p5 => prompt_buffer'address,
               p6 => prompt_buffer'length);
END vms_routine;
```

15.2 Implementation-Dependent Attributes

The name, type and implementation-dependent aspects of every implementation-dependent attribute is stated in this section.

15.2.1 Language-Defined Attributes

The name and type of all the language-defined attributes are as given in the LRM. We note here only the implementation-dependent aspects.

ADDRESS

If this attribute is applied to an object for which storage is allocated, it yields the address of the first storage unit that is occupied by the object.
If it is applied to a subprogram or to a task, it yields the address of the entry point of the subprogram or task body.
If it is applied to a task entry for which an address clause is given, it yields the address given in the address clause.
For any other entity this attribute is not supported and will return the value `system.address_zero`.

IMAGE

The image of a character other than a graphic character (cf. LRM(§3.5.5(11))) is the string obtained by replacing each italic character in the indication of the character literal (given in the LRM(Annex C(13))) by the corresponding upper-case character. For example, `character'image(`*nul*`) = "NUL"`.

MACHINE_OVERFLOWS

Yields true for each real type or subtype.

MACHINE_ROUNDS

Yields true for each real type or subtype.

STORAGE_SIZE

The value delivered by this attribute applied to an access type is as follows:
If a length specification (STORAGE_SIZE, see §16.2) has been given for that type (static collection), the attribute delivers that specified value.
In case of a dynamic collection, i.e. no length specification by STORAGE_SIZE given for the access type, the attribute delivers the number of storage units currently allocated for the collection. Note that dynamic collections are extended if needed.
If the collection manager (cf. §13.3.1) is used for a dynamic collection the attribute delivers the number of storage units currently allocated for the collection. Note that in this case the number of storage units currently allocated may be decreased by release operations.

The value delivered by this attribute applied to a task type or task object is as follows:
If a length specification (STORAGE_SIZE, see §16.2) has been given for the task type, the attribute delivers that specified value; otherwise, the default value is returned.

15.2.2 Implementation-Defined Attributes

There are no implementation-defined attributes.

15.3 Specification of the Package SYSTEM

The package system as required in the LRM(§13.7) is reprinted here with all implementation-dependent characteristics and extensions filled in.

```
PACKAGE system IS

   TYPE designated_by_address IS LIMITED PRIVATE;

   TYPE address IS ACCESS designated_by_address;
   FOR address'storage_size USE 0;

   address_zero : CONSTANT address := NULL;

   TYPE name IS (vax_vms);

   system_name  : CONSTANT name := vax_vms;

   storage_unit : CONSTANT := 8;
   memory_size  : CONSTANT := 2 ** 31;
   min_int      : CONSTANT := - 2 ** 31;
   max_int      : CONSTANT := 2 ** 31 - 1;
   max_digits   : CONSTANT := 33;
   max_mantissa : CONSTANT := 31;
   fine_delta   : CONSTANT := 2.0 ** (-31);
   tick         : CONSTANT := 0.01;

   SUBTYPE priority IS integer RANGE 0 .. 15;

   FUNCTION "+" (left : address; right : integer) RETURN address;

   FUNCTION "+" (left : integer; right : address) RETURN address;

   FUNCTION "-" (left : address; right : integer) RETURN address;

   FUNCTION "-" (left : address; right : address) RETURN integer;

   SUBTYPE external_address IS STRING;

   -- External addresses use hexadecimal notation with characters
   -- '0'..'9', 'a'..'f' and 'A'..'F'. For instance:
   --    "7FFFFFFF"
   --    "80000000"
```

```
--    "8" represents the same address as "00000008"

FUNCTION convert_address (addr : external_address) RETURN address;

   -- CONSTRAINT_ERROR is raised if the external address ADDR
   -- is the empty string, contains characters other than
   -- '0'..'9', 'a'..'f', 'A'..'F' or if the resulting address
   -- value cannot be represented with 32 bits.

FUNCTION convert_address (addr : address) RETURN external_address;

   -- The resulting external address consists of exactly 8
   -- characters '0'..'9', 'A'..'F'.

TYPE interrupt_number IS RANGE 0 .. 31;

TYPE interrupt_addresses IS ARRAY (interrupt_number) OF address;

ast_service,
interrupt_vector : interrupt_addresses;

non_ada_error       : EXCEPTION;

-- non_ada_error is raised, if some event occurs which does not
-- correspond to any situation covered by Ada, e.g.:
--    illegal instruction encountered
--    error during address translation
--    illegal address

TYPE exception_id IS NEW address;

no_exception_id     : CONSTANT exception_id := address_zero;

-- Coding of the predefined exceptions:

constraint_error_id : CONSTANT exception_id := ... ;
numeric_error_id    : CONSTANT exception_id := ... ;
program_error_id    : CONSTANT exception_id := ... ;
storage_error_id    : CONSTANT exception_id := ... ;
tasking_error_id    : CONSTANT exception_id := ... ;

non_ada_error_id    : CONSTANT exception_id := ... ;

status_error_id     : CONSTANT exception_id := ... ;
mode_error_id       : CONSTANT exception_id := ... ;
name_error_id       : CONSTANT exception_id := ... ;
use_error_id        : CONSTANT exception_id := ... ;
```

```
device_error_id      : CONSTANT exception_id := ... ;
end_error_id         : CONSTANT exception_id := ... ;
data_error_id        : CONSTANT exception_id := ... ;
layout_error_id      : CONSTANT exception_id := ... ;

time_error_id        : CONSTANT exception_id := ... ;

TYPE argument_array IS ARRAY (1 .. 4) OF integer;

no_condition_name    : CONSTANT := 1;

TYPE exception_information IS
   RECORD
      excp_id           : exception_id;

         -- Identification of the exception. The codings of
         -- the predefined exceptions are given above.

      code_addr         : address;

         -- Code address where the exception occurred. Depending
         -- on the kind of the exception it may be be address of
         -- the instruction which caused the exception, or it
         -- may be the address of the instruction which would
         -- have been executed if the exception had not occurred.

      condition_name    : integer;

         -- If /= no_condition_name, the exception was caused
         -- by a condition. In this case, the condition name
         -- and other following information made available.

      nr_of_arguments   : integer;   -- in the range 1 .. 4.

      arguments         : argument_array;

         -- Only arguments (1 .. nr_of_arguments) are valid.
         -- It contains a copy of the optional information
         -- supplied by VMS in the argument array when the
         -- condition occurred. If there are more than 4 optional
         -- entries in the argument array, only the first 4
         -- are copied.

      psl               : integer;

         -- The processor status longword.
```

```
      END RECORD;

   PROCEDURE get_exception_information
               (excp_info : OUT exception_information);

      -- The subprogram get_exception_information must only be called
      -- from within an exception handler BEFORE ANY OTHER EXCEPTION
      -- IS RAISED. It then returns the information record about the
      -- actually handled exception.
      -- Otherwise, its result is undefined.

   TYPE exit_code IS NEW integer;

   -- The exit codes WARNING, ERROR and SEVERE_ERROR set the bit 28,
   -- which inhibits the display of the error message 0 by the DCL
   -- interpreter
   warning       : CONSTANT exit_code := 16#10000000#;
   success       : CONSTANT exit_code := 16#00000001#;
   error         : CONSTANT exit_code := 16#10000002#;
   information   : CONSTANT exit_code := 16#00000003#;
   severe_error : CONSTANT exit_code := 16#10000004#;
   PROCEDURE set_exit_code (val : exit_code);

      -- Specifies the exit code which is returned to the
      -- operating system if the Ada program terminates normally.
      -- The default exit code is 'success'. If the program is
      -- abandoned because of an exception, the exit code is
      -- 'error'.

PRIVATE

   -- private declarations

END system;
```

15.4 Restrictions on Representation Clauses

See Chapter 16 of this manual.

15.5 Conventions for Implementation-Generated Names

There are implementation generated components but these have no names. (cf. §16.4 of this manual).

15.6 Expressions in Address Clauses

See §16.5 of this manual.

15.7 Restrictions on Unchecked Conversions

The implementation supports unchecked type conversions for all kinds of source and target types with the restriction that the target type must not be an unconstrained array type. The result value of the unchecked conversion is unpredictable, if

```
target_type'SIZE > source_type'SIZE
```

15.8 Characteristics of the Input-Output Packages

The implementation-dependent characteristics of the input-output packages as defined in the LRM(Chapter 14) are reported in Chapter 17 of this manual.

15.9 Requirements for a Main Program

A main program must be a parameterless library procedure. This procedure may be a generic instantiation; the generic procedure need not be a library unit.

15.10 Unchecked Storage Deallocation

The generic procedure `unchecked_deallocation` is provided; the effect of calling an instance of this procedure is as described in the LRM(§13.10.1).

The implementation also provides an implementation-defined package `collection_manager`, which has advantages over unchecked deallocation in some applications (cf. §13.3.1).

Unchecked deallocation and operations of the `collection_manager` can be combined as follows:

- `collection_manager.reset` can be applied to a collection on which unchecked deallocation has also been used. The effect is that storage of all objects of the collection is reclaimed.
- After the first `unchecked_deallocation` (`release`) on a collection, all following calls of `release` (`unchecked deallocation`) until the next `reset` have no effect, i.e. storage is not reclaimed.
- after a `reset` a collection can be managed by `mark` and `release` (resp. `unchecked_deallocation`) with the normal effect even if it was managed by `unchecked_deallocation` (resp. `mark` and `release`) before the `reset`.

15.11 Machine Code Insertions

A package `machine_code` is not provided and machine code insertions are not supported.

15.12 Numeric Error

The predefined exception `numeric_error` is never raised implicitly by any predefined operation; instead the predefined exception `constraint_error` is raised.